Program Library HOWTO

David A. Wheeler

version 1.20, 11 April 2003

This HOWTO for programmers discusses how to create and use program libraries on Linux. This includes static libraries, shared libraries, and dynamically loaded libraries.

---

4. Dynamically Loaded (DL) Libraries

Dynamically loaded (DL) libraries are libraries that are loaded at times other than during the startup of a program. They're particularly useful for implementing plugins or modules, because they permit waiting to load the plugin until it's needed. For example, the Pluggable Authentication Modules (PAM) system uses DL libraries to permit administrators to configure and reconfigure authentication. They're also useful for implementing interpreters that wish to occasionally compile their code into machine code and use the compiled version for efficiency purposes, all without stopping. For example, this approach can be useful in implementing a just-in-time compiler or multi-user dungeon (MUD).

In Linux, DL libraries aren't actually special from the point-of-view of their format; they are built as standard object files or standard shared libraries as discussed above. The main difference is that the libraries aren't automatically loaded at program link time or start-up; instead, there is an API for opening a library, looking up symbols, handling errors, and closing the library. C users will need to include the header file <dlfcn.h> to use this API.

The interface used by Linux is essentially the same as that used in Solaris, which I'll call the ``dlopen()'' API. However, this same interface is not supported by all platforms; HP-UX uses the different shl_load() mechanism, and Windows platforms use DLLs with a completely different interface. If your goal is wide portability, you probably ought to consider using some wrapping library that hides differences between platforms. One approach is the glib library with its support for Dynamic Loading of Modules; it uses the underlying dynamic loading routines of the platform to implement a portable interface to these functions. You can learn more about glib at http://developer.gnome.org/doc/API/glib/glib-dynamic-loading-of-modules.html. Since the glib interface is well-explained in its documentation, I won't discuss it further here. Another approach is to use libltdl, which is part of GNU libtool. If you want much more functionality than this, you might want to look into a CORBA Object Request Broker (ORB). If you're still interested in directly using the interface supported by Linux and Solaris, read on.

Developers using C++ and dynamically loaded (DL) libraries should also consult the ``C++ dlopen mini-HOWTO''.

4.1. dlopen()

The dlopen(3) function opens a library and prepares it for use. In C its prototype is:
```
void * dlopen(const char *filename, int flag);
```
If filename begins with ``/'' (i.e., it's an absolute path), dlopen() will just try to use it (it won't search for a library). Otherwise, dlopen() will search for the library in the following order:

1. A colon-separated list of directories in the user's LD_LIBRARY_PATH environment variable.

2. The list of libraries specified in /etc/ld.so.cache (which is generated from /etc/ld.so.conf).

3. /lib, followed by /usr/lib. Note the order here; this is the reverse of the order used by the old a.out loader. The old a.out loader, when loading a program, first searched /usr/lib, then /lib (see the man page ld.so(8)). This shouldn't normally matter, since a library should only be in one or the other directory (never both), and different libraries with the same name are a disaster waiting to happen.

In dlopen(), the value of *flag* must be either RTLD_LAZY, meaning ``resolve undefined symbols as code from the dynamic library is executed'', or RTLD_NOW, meaning ``resolve all undefined symbols before dlopen() returns and fail if this cannot be done''. RTLD_GLOBAL may be optionally or'ed with either value in *flag*, meaning that the external symbols defined in the library will be made available to subsequently loaded libraries. While you're

debugging, you'll probably want to use RTLD_NOW; using RTLD_LAZY can create inscrutable errors if there are unresolved references. Using RTLD_NOW makes opening the library take slightly longer (but it speeds up lookups later); if this causes a user interface problem you can switch to RTLD_LAZY later.

If the libraries depend on each other (e.g., X depends on Y), then you need to load the dependees first (in this example, load Y first, and then X).

The return value of dlopen() is a ``handle'' that should be considered an opaque value to be used by the other DL library routines. dlopen() will return NULL if the attempt to load does not succeed, and you need to check for this. If the same library is loaded more than once with dlopen(), the same file handle is returned.

In older systems, if the library exports a routine named _init, then that code is executed before dlopen() returns. You can use this fact in your own libraries to implement initialization routines. However, libraries should not export routines named _init or _fini. Those mechanisms are obsolete, and may result in undesired behavior. Instead, libraries should export routines using the __attribute__((constructor)) and __attribute__((destructor)) function attributes (presuming you're using gcc). See Section 5.2 for more information.

4.2. dlerror()

Errors can be reported by calling dlerror(), which returns a string describing the error from the last call to dlopen(), dlsym(), or dlclose(). One oddity is that after calling dlerror(), future calls to dlerror() will return NULL until another error has been encountered.

4.3. dlsym()

There's no point in loading a DL library if you can't use it. The main routine for using a DL library is dlsym(3), which looks up the value of a symbol in a given (opened) library. This function is defined as:

```
void * dlsym(void *handle, char *symbol);
```

the handle is the value returned from dlopen, and symbol is a NIL-terminated string. If you can avoid it, don't store the result of dlsym() into a void* pointer, because then you'll have to cast it each time you use it (and you'll give less information to other people trying to maintain the program).

dlsym() will return a NULL result if the symbol wasn't found. If you know that the symbol could never have the value of NULL or zero, that may be fine, but there's a potential ambiguity otherwise: if you got a NULL, does that mean there is no such symbol, or that NULL is the value of the symbol? The standard solution is to call dlerror() first (to clear any error condition that may have existed), then call dlsym() to request a symbol, then call dlerror() again to see if an error occurred. A code snippet would look like this:

```
dlerror(); /* clear error code */
s = (actual_type) dlsym(handle, symbol_being_searched_for);
if ((err = dlerror()) != NULL) {
 /* handle error, the symbol wasn't found */
} else {
 /* symbol found, its value is in s */
}
```

4.4. dlclose()

The converse of dlopen() is dlclose(), which closes a DL library. The dl library maintains link counts for dynamic file handles, so a dynamic library is not actually deallocated until dlclose has been called on it as many times as dlopen has succeeded on it. Thus, it's not a problem for the same program to load the same library multiple times. If a library is deallocated, its function _fini is called (if it exists) in older libraries, but _fini is an obsolete mechanism and shouldn't be relied on. Instead, libraries should export routines using the __attribute__((constructor)) and __attribute__((destructor)) function attributes. See Section 5.2 for more information. Note: dlclose() returns 0 on success, and non-

.zero on error; some Linux manual pages don't mention this.

4.5. DL Library Example

Here's an example from the man page of dlopen(3). This example loads the math library and prints the cosine of 2.0, and it checks for errors at every step (recommended):

```
#include <stdlib.h>
#include <stdio.h>
#include <dlfcn.h>

int main(int argc, char **argv) {
    void *handle;
    double (*cosine)(double);
    char *error;

    handle = dlopen ("/lib/libm.so.6", RTLD_LAZY);
    if (!handle) {
        fputs (dlerror(), stderr);
        exit(1);
    }

    cosine = dlsym(handle, "cos");
    if ((error = dlerror()) != NULL)  {
        fputs(error, stderr);
        exit(1);
    }

    printf ("%f\n", (*cosine)(2.0));
    dlclose(handle);
}
```

If this program were in a file named "foo.c", you would build the program with the following command:

```
gcc -o foo foo.c -ldl
```